IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 21, 2018, accepted July 22, 2018, date of publication August 7, 2018, date of current version August 28, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2864253

Revisiting Genetic Network Programming (GNP): Towards the Simplified Genetic Operators

XIANNENG LI, HUIYAN YANG, AND MEIHUA YANG
Faculty of Management and Economics, Dalian University of Technology, Dalian 116024, China

Corresponding author: Xianneng Li (xianneng@dlut.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 71601028, Grant 71671024, Grant 71421001, and Grant 71431002, in part by the Fundamental Research Funds for the Central Universities under Grant DUT17JC12, and in part by the Economic & Social Development Foundation of Liaoning under Grant 2018lslktqn-015.

ABSTRACT Genetic network programming (GNP) is a relatively new type of graph-based evolutionary algorithm, which designs a directed graph structure for its individual representation. A number of studies have demonstrated its expressive ability to model complicated problems/systems and explored it from the perspectives of methodologies and applications. However, the unique features of its directed graph are relatively unexplored, which cause unnecessary dilemma for the further usage and promotion. This paper is dedicated to uncover this issue systematically and theoretically. It is proved that the traditional GNP with uniform genetic operators does not consider the ''transition by necessity'' feature of the directed graph, which brings the unnecessary difficulty of evolution to cause invalid/negative evolution problems. Consequently, simplified genetic operators are developed to address these problems. Experimental results on two benchmark testbeds of the agent control problems are carried out to demonstrate its superiority over the traditional GNP and the state-of-the-art algorithms in terms of fitness results, search speed, and computation time.

INDEX TERMS Directed graph, evolutionary algorithms, genetic network programming, invalid/negative evolution, transition by necessity.

I. INTRODUCTION

Evolutionary algorithms (EAs) are a family of optimization techniques which take inspirations from the theory of biological evolution via natural selection and genetic operations. Over the past decades, numerous EAs have been developed to solve different types of optimization problems. In addition to the original inspirations of different algorithms, one of the most important features to distinguish each EA from one another is the individual representation (also known as the chromosome structure), hence different problem types fitted.

The typical representatives of EAs include genetic algorithm (GA) [1], evolution strategy (ES) [2], particle swarm optimizer (PSO) [3], differential evolution (DE) [4], and genetic programming (GP) [5], etc. GA is initially designed to evolve bit-string individuals [1], and later extends to real-coded variants [6]. ES, PSO, and DE represent their individuals by real-coded variables to directly optimize the continuous variables. Genetic programming (GP) [5], as another typical example, designs trees as its individual representation. It explores EAs for the automatic evolution of computer programs, consequently allowing modeling more complicated problems. Different EAs are generally suitable for their own particular problems, which cannot guarantee the superiority in all problems, so called No-Free-Lunch (NFL) theorem [7]. Therefore, there has always been interest to develop new algorithms when focusing on some particular sorts of problems. Although it is theoretically impossible to have a best, general, and universal optimization technique, with prior knowledge one can develop a particular algorithm to suit specific classes of optimization problems.

Following the NFL theorem, one successful path of exploring EAs is extending the individual representation from bit-strings, real-coded variables, and trees to graphical structures [8]–[10]. Unlike the other structures, graphs allow to model more complicated problems through the arbitrary connections among nodes. Among different graphical EAs, we focus on a relatively novel one, named genetic network programming (GNP) [9], [11]. In GNP, a unique directed graph is designed to represent its individual, which consists of two types of nodes: *judgment* node and *processing* node. The judgment/processing nodes constitute the unique directed graph structure to efficiently and flexibly generate compact programs, which derive a sequence of ''IF-THEN'' decision-making rules.

GNP has been successfully applied to various sorts of problems, such as agent control [11], [12], elevator control [13], robot control [14], stock trading [15], and data mining [16], etc, along with many methodological enhancements. However, the systematic and theoretical study of GNP with respect to its unique directed graph is unexplored in the literature, which remains the major gap to support its further usage and promotion. Most importantly, when the distinguished features of GNP's directed graph are not taken into account, the unnecessary difficulty of evolution will appear, especially when using the traditional genetic operators.[1]

To address the above issues, an in-depth discussion of GNP is provided in this paper, so that its unique features and advantages become clear. By a systematic and theoretical study of the directed graph, "*transition by necessity*" feature of GNP is presented. Ignoring such a feature, it is proved that traditional genetic operators tend to cause the serious drawbacks of evolution, named *invalid evolution* and *negative evolution*. To solve them, novel evolution strategies — simplified genetic operators — are proposed, which explicitly take the above feature into account. The proposed genetic operators are capable of relaxing the negative effect of invalid/negative evolution and reducing the search dimensions of GNP, which consequently provide better evolution results with faster search speed. This work is a significant extension of a conference version [18]. The major contributions of this work differing from reference [18] include the following parts: 1) We reveal GNP's unique directed graphs systematically and theoretically to demonstrate its distinguished "transition by necessity" feature; 2) The invalid evolution and negative evolution problems caused by uniform genetic operators are theoretically proved and measured, where the early work [18] only presented a descriptive study; 3) Simplified genetic operators are enhanced with slight modifications of [18] to take the "transition by necessity" feature into account more appropriately (details refer to section III); 4) The theoretical study of simplified genetic operators is provided; 5) The experimental studies are significantly enhanced via two benchmark testbeds of the agent control problems, i.e., maze problems [19] and Tileworld [20].

The paper is organized as follows. GNP, its evolution, and its drawbacks are revisited in section II. The proposed simplified genetic operators are described in section III. Section IV presents the experimental studies in two benchmark testbeds of the agent control problems, i.e., maze problems [19] and Tileworld [20]. Finally, the conclusions are drawn.

II. REVISITING GENETIC NETWORK PROGRAMMING (GNP)

A. INDIVIDUAL REPRESENTATION: DIRECTED GRAPH

GNP designs a unique structure — directed graph — for its individual representation. In the directed graph, two types of nodes are defined:

- *Judgment node*: It plays the role of judging the information from the environments.
- *Processing node*: It plays the role of processing actions in accordance to the results of its antecedent judgments.

Each judgment node consists of a judgment function, which deals with a specific input of the problems, i.e., the sensory results in robot control. Each processing node enforces a processing/action function, such as the movement speed or direction in robot control. In general, GNP represents the directed graph by symbol G as follows.

$$G = (S_{\text{node}}, \text{LIBRARY}), \tag{1}$$

where,
S_{node}: the set of nodes included in G;
LIBRARY: the set of judgment/processing functions defined by the problems.

For a specific node $i \in S_{\text{node}}$, it is represented as follows.

$$i = (NT_i, NF_i, B(i), C_i), \tag{2}$$

$$C_i = \{C_{i1}, C_{i2}, \ldots, C_{iL_i}\}, \tag{3}$$

where,
NT_i: the node type (judgment or processing node);
NF_i: the node function which is defined in LIBRARY;
$B(i)$: the set of branches;
C_i: the connection information, represented by a set $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iL_i}\}$;
C_{ik}: the node connected from the k_{th} $(1 \leq k \leq L_i)$ branch of node i;
L_i: the total number of branches in node i, and $L_i = |B(i)|$.

The judgment nodes have multiple branches, referring to all the possible judgment results, while the processing nodes only consist of one branch (means "no conditional branch"). Therefore, the size of $B(i)$ is equal to 1 for processing nodes ($L_i = 1$) or a value larger than 1 for judgment nodes ($L_i > 1$).

When GNP is applied to solve a problem, the number of judgment/processing nodes is pre-designed by users (usually by the hand-tuning strategy). During evolution, the node type, node function, and number of branches per each node are unchangeable. In other words, GNP evolution only endorses the *node connections* to be evolved. That is, only the connection information C_i of each node i is evolved. Accordingly, we have the search dimensions (reflecting the search space) of GNP as follows.

Definition 1: Given a directed graph g of GNP, the set of search dimensions D is defined by the total number of branches, i.e., $D = \bigcup_{i \in S_{\text{node}}} B(i)$. Since $L_i = |B(i)|$, we have $|D| = \sum_{i \in S_{\text{node}}} L_i$.

In Figure 1, we illustrate a typical example for a more detailed analysis. It includes the phenotype expression (directed graph structure) and genotype expression (bit-string gene structure). The most distinguished features of GNP arise from this directed graph, which can be summarized as follows.

[1] Uniform genetic operators are considered in this paper, since it is the standard and most-widely used option in GNP literature [11], [17].

- As the directed graph is designed with a fixed number of nodes ($|S_{\text{node}}|$ unchanged), the bloat problem of GP will never happen in GNP.
- As the nodes can be connected arbitrary in the directed graph, evolution power will gradually encourage the reuse of useful/important nodes to formulate the module-like repetitive transitions/processes.
- Evolution is restricted within the node connections, so that the compact programs can be efficiently generated and evolved through a relatively small size of directed graph.

*Example 1 (**Directed Graph of GNP**):* In Figure 1, the directed graph G of GNP consists of $|S_{\text{node}}| = 6$ nodes. The LIBRARY has 4 judgment functions (J_1, J_2, J_3 and J_4) and 2 processing functions (P_1 and P_2). Each judgment node i consists of 2 branches ($i(1)$ and $i(2)$). The detailed node information is listed in the right figure, where evolution is applied to evolve the value of C_i for each node i. It can be explicitly found that with the change of C_i, new node connections can be formed, and the directed graph is changed consequently. The number of search dimensions of this directed graph are $|D| = 4 \times 2 + 2 \times 1 = 10$ (the total number of branches).

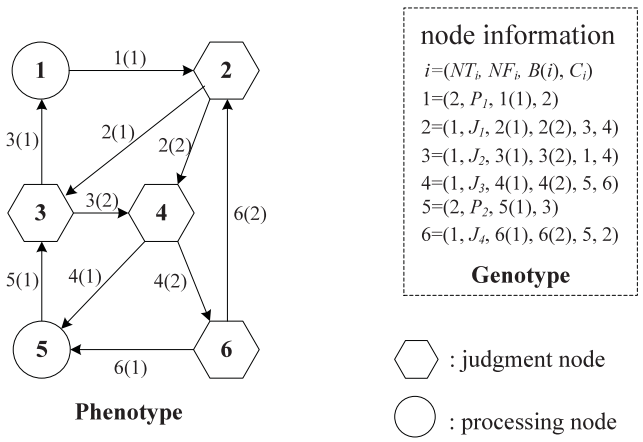

FIGURE 1. Directed graph of GNP. Left: Phenotype expression (directed graph structure); Right: Genotype expression (bit-string gene structure)

B. COMPARISON WITH THE EXISTING GRAPH-BASED EA

GNP distinguishes itself from the large amount of EAs by the above unique directed graph structure. Naturally, the directed graph ensures higher expression ability and more flexibility to potentially model more complicated problems than traditional EAs, such as GA, DE, and GP, etc. Notable results have been found in various problems [11], [12], [21]. In addition to GNP's directed graph, various graph-based EAs have been proposed in the literature [8], [22]–[24]. The main features of GNP differing from the other graph-based EAs are as follows.

- There is no terminal node in GNP, where the GNP programs terminate once the task is solved.
- GNP separates the nodes into judgment/processing nodes, so that the "IF(*necessary* judgments), THEN (processing)" [2] type decision-making rules can be easily formulated through the node transitions of the directed graph.
- With the above forms, GNP is particularly suitable to model the *behavior of intelligent agents*, deriving numerous real-world applications.

C. TRANSITION BY NECESSITY

When applying the directed graph to solve problems, the solutions are formulated as compact programs through the node transitions of the directed graph. More precisely, the node transitions can efficiently generate the sequential "IF(necessary judgments), THEN(processing)" type decision-making rules, which play the essential role for problem solving. In each rule, only some of the judgment functions are made (necessary judgments) to make the action determination (processing), where the necessity of judgment is determined by evolution. With the above discussion, we introduce the most important feature of GNP's directed graph — "*transition by necessity*". It is derived from the functional distribution of neuroscience, that human brain usually activates a particular part of the neurons when perceiving specific inputs [25]. Figure 2 illustrates a typical example of problem solving by GNP's directed graph through its node transitions.

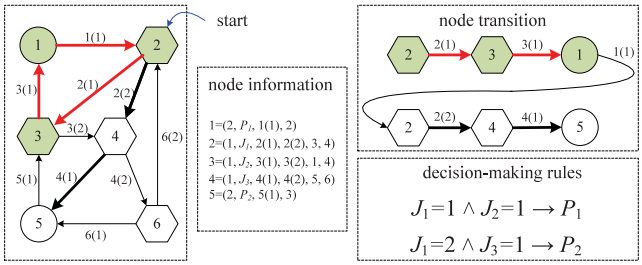

FIGURE 2. Problem solving by GNP's directed graph (details in Example 2).

*Example 2 (**Problem Solving by GNP**):* In the example of Figure 2, the target problem is a two-step problems. The GNP program starts from node 2, where the hereafter nodes are transited in accordance to the interaction with the problem environments. For each judgment node (with multiple branches), the next transited node is determined by the judgment results. By following the node transitions, a sequence of "IF(necessary judgments), THEN(processing)" is generated to form the compact programs of GNP, which can be derived to the expressive decision-making rules. As shown in the figure, two decision-making rules are generated to solve the two-step problems.

In Example 2, only parts of the directed graph are transited (node 6 is not used). Some important nodes are frequently transited (node 2 is transited two times). Two decision-making rules are consequently generated, within which only

[2]The word "*necessary*" indicates that not all judgment functions should be judged.

Algorithm 1: Procedure of GNP

```
define the directed graph structure (N_node, LIBRARY);
define population size |Pop|, sets of elite, crossover, and
  mutation individuals (Pop_e, Pop_c and Pop_m);
initialize population Pop;
repeat
    preserve the best |Pop_e| individuals (elite selection);
    for index i = 0; i < |Pop_c|/2; i++ do
        execute uniform crossover in Pop_c;
    for index i = 0; i < |Pop_m|; i++ do
        execute uniform mutation in Pop_m;
until terminal conditions;
```

two judgments are made for processing. This explicitly demonstrates the *generalization ability* of GNP over conventional algorithms which take all judgments into account. We further present an experiment in Tileworld testbed (problem description in section IV) in Figure 3. In this case, there are total 220 branches in the directed graph, where only less than 160 branches are transited. These two examples essentially describe the "transition by necessity" feature of GNP.

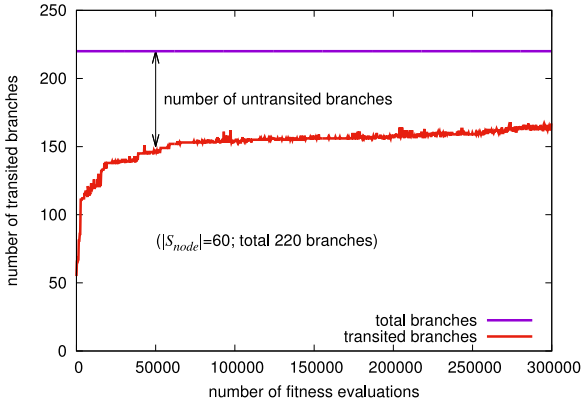

FIGURE 3. "Transition by necessity" of GNP (in Tileworld with $ST = 60$).

D. EVOLUTIONARY MECHANISM OF GNP

The procedure of GNP is shown in Algorithm 1. As the other EAs, GNP (as well as its variants) iteratively evolves a population *Pop* of directed graphs which are initialized randomly. In each generation, *Pop* is divided into three groups: elite, crossover and mutation individuals (Pop_e, Pop_c and Pop_m, respectively). They are subject to evolution by elite selection and genetic operators (crossover and mutation). Any type of standard genetic operators can be used, such as one-point, two-points, and uniform genetic operators, etc. However, previous studies demonstrated that uniform genetic operators are more favorable for GNP evolution than the others [17]. Therefore, they have become the mostly used option in GNP literature [11].

In uniform crossover, each branch of the parents is selected with crossover rate p_c, where the node connections are exchanged with each other. Uniform mutation selects each branch with mutation rate p_m to randomly change its node connections. The typical examples of uniform genetic operators are illustrated in Figure 4, including uniform crossover and uniform mutation.

*Example 3 (**Uniform Crossover of GNP**):* In the left part of Figure 4, Parent individuals P1 and P2 are selected for uniform crossover, and each branch has equal probability p_c for crossover action. Here, the second branch of node 3, i.e., branch 3(2), is selected for crossover. The corresponding node connections ($C_{32} = 4$ in P1 and $C_{32} = 5$ in P2) are exchanged with each other to generate two Offspring O1 and O2.

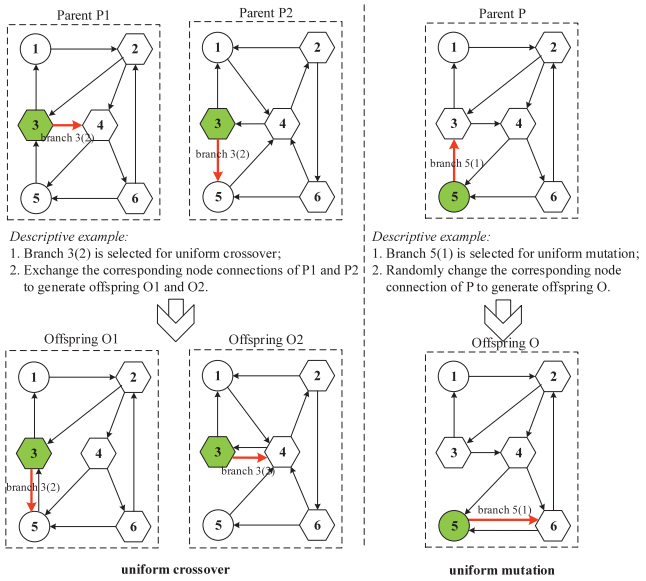

FIGURE 4. Uniform genetic operators of GNP.

*Example 4 (**Uniform Mutation of GNP**):* In the right part of Figure 4, Parent P is selected for mutation, and each branch has equal probability p_m for mutation action. In this example, the first branch of node 5, i.e., branch 5(1), in P is selected for mutation. The corresponding node connection ($C_{51} = 3$ in P) is randomly changed to connect to another node, i.e., $C_{51} = 6$, to generate offspring O.

E. PROBLEM DESCRIPTION

Uniform genetic operators (as well as the others) do not consider the "transition by necessity" feature of GNP at all. This provides the advantage of easy implementation, however, which tends to cause the serious drawbacks of evolution, potentially deteriorating the evolution efficiency. In this paper, we address them by deriving two problems, named *invalid evolution* and *negative evolution*, which are discussed hereafter.

1) INVALID EVOLUTION PROBLEM

Given a directed graph (individual) $g \in Pop$, after its execution, we have two sets of branches: transited branches (TB_g) and untransited branches (UB_g). These two sets satisfy

$$TB_g \cap UB_g = \emptyset; \tag{4}$$
$$TB_g \cup UB_g = D. \tag{5}$$

Supposing the fitness result of g are denoted as $f(g)$, based on the "transition by necessity" feature, $f(g)$ is only influenced by the evaluation of TB_g (the set of transited branches). UB_g (the set of untransited branches) has not any effect on the calculation of $f(g)$. Ideally, evolving UB_g or not does not cause any reasonable and theoretical influence of $f(g)$. However, in uniform genetic operators, all branches have equivalent opportunity to be evolved, regardless the fact that evolving UB_g is meaningless, causing a serious problem – "*Invalid evolution*".

Definition 2: "*Invalid evolution*" problem is defined by the phenomenon of evolving the untransited branches in each generation.

To present a quantitative study, this problem can be measured by a metric named *invalid evolution rate* (*IER*).

Definition 3: *IER* is defined by the *rate* that the untransited branches are evolved over the entire evolved branches in each generation.

With the above definition, we can analyze the *IER* of uniform genetic operators, i.e., *IER*(*uniform*), as follows.

Given a uniform crossover individual g, the theoretical number of untransited branches to be evolved is $|UB_g| \times p_c$, and that of a uniform mutation individual g is $|UB_g| \times p_m$. If we define the total number of untransited branches to be evolved in each generation as *evol_UB*, it can be calculated as follows.

$$evol\_UB = \sum_{g \in Pop_c} |UB_g| \times p_c + \sum_{g \in Pop_m} |UB_g| \times p_m. \tag{6}$$

For each uniform crossover individual, the theoretical number of evolved branches is $p_c \times \sum_{i \in S_{\text{node}}} L_i$, and that of each uniform mutation individual is $p_m \times \sum_{i \in S_{\text{node}}} L_i$. Multiplied by the number of crossover/mutation individuals, we have the total number of evolved branches, i.e., *evol*, as follows.

$$\begin{aligned} evol &= |Pop_c| \times p_c \times \sum_{i \in S_{node}} L_i + |Pop_m| \times p_m \times \sum_{i \in S_{node}} L_i \\ &= (|Pop_c| \times p_c + |Pop_m| \times p_m) \times \sum_{i \in S_{node}} L_i. \end{aligned} \tag{7}$$

According to Definition 3, we can obtain *IER*(*uniform*) of each generation as follows.

$$IER(uniform) = \frac{evol\_UB}{evol} \times 100\%. \tag{8}$$

With the increase of unused sub-graph region (untransited branches *UB*), *IER*(*uniform*) tends to be increased, which indicates that uniform genetic operators generally fall into meaningless evolution of GNP. Removing these evolution actions would not have any negative influence of the evolution performance.

2) NEGATIVE EVOLUTION PROBLEM

Invalid evolution problem is carried out when we analyze the evolution behavior of a *single* generation. In this section, we relax the analysis to *multiple* generations to propose the second problem caused by uniform genetic operators — "*negative evolution*".

We consider two different generations, i.e., generation $t1$ and $t2$ ($t1 < t2$). For each directed graph g, the sets of transited/untransited branches generally vary in generation $t1$ and $t2$, that,[3]

$$TB_g(t1) \neq TB_g(t2), \quad \text{and} \quad UB_g(t1) \neq UB_g(t2). \tag{9}$$

Suppose we have a specific branch b of directed graph g, which satisfies the following three conditions:

1) $b \in TB_g(t1)$: b is transited (used) in an early generation $t1$;
2) $b \in UB_g(t2)$: b is not transited (used) in a later generation $t2$ ($> t1$);
3) b is not evolved from generation $t1$ to $t2$ (b remains unchanged until $t2$).

Although b is not used in generation $t2$, it was previously used in the early generation $t1$ and remained unevolved until $t2$. This reflects the necessity of node transitions that b can adapt some particularly perceived environments. In other words, the node connection of branch b represents a sort of *building-blocks* (BBs) [4] in the early generation $t1$. This information (b) is remained unchanged (unevolved) during $t1$ and $t2$, meaning that the corresponding BBs are preserved.

If we consider to evolve b in generation $t2$, invalid evolution problem will be caused. This is obvious since b is unused in generation $t2$. Evolving b in generation $t2$ is meaningless, since this branch is not used (and evaluated) in this generation. Meantime, this evolution action can also break the BBs obtained in the previous generation $t1$ without evaluation. If many branches like b are evolved in this way, the frequent breakage of BBs will be caused seriously. This significantly increases the uncertainty of evolution, since the evolution and evaluation are not synchronized. The frequent breakage of BBs will lower the evolution efficiency, and eventually deteriorate the generalization ability of the directed graph (since worse final solution is obtained and the high-quality subgraphs, i.e., BBs, are easily destroyed without evaluation).

In short, the brute-force evolution of uniform genetic operators in GNP can easily break the BBs without evaluation ("Without evaluation" will make us hard to judge this evolution is good or not), and we name this problem via negative evolution.

[3]This phenomenon is normal in GNP execution, and it can be easily explained by the "transition by necessity" feature that different sub-graphs are activated to suit the particular scenarios of different generations.

[4]Building-blocks (BBs) refer to the partial solutions with high quality [1].

Definition 4: "*Negative evolution*" problem is defined by the phenomenon of evolving the untransited branches which were transited in the previous generations.

This problem can be quantitatively studied by measuring the *negative evolution rate* (*NER*) metric.

Definition 5: In each generation, supposing the set of untransited branches which were transited in the previous generations but remained unevolved until now is denoted as *NB* (branches causing negative evolution, and $NB \subseteq UB$), *NER* is defined by the *rate* that the branches of *NB* are evolved over the entire evolved branches.

With the above definition, we can analyze the *NER* of uniform genetic operators, i.e., *NER*(*uniform*), as follows, which is similar to *IER*(*uniform*).

Given a uniform crossover individual g, the theoretical number of evolved branches causing negative evolution is $|NB_g| \times p_c$, and that of a uniform mutation individual g is $|NB_g| \times p_m$. If we define the total number of evolved branches causing negative evolution in each generation as *evol_NB*, it can be calculated as follows.

$$evol\_NB = \sum_{g \in Pop_c} |NB_g| \times p_c + \sum_{g \in Pop_m} |NB_g| \times p_m. \quad (10)$$

Dividing *evol_NB* by the total number of evolved branches, i.e., *evol* of Equation (7), we can obtain *NER*(*uniform*) of each generation based on Definition 5.

$$NER(uniform) = \frac{evol\_NB}{evol} \times 100\%. \quad (11)$$

Since $NB \subseteq UB$, *NER*(*uniform*) is generally smaller than *IER*(*uniform*) in each generation. In other words, negative evolution has relatively smaller effect than invalid evolution when using uniform genetic operators. However, it will cause worse effect to deteriorate the evolution performance than invalid evolution. The reason is as follows: invalid evolution will only increase the unnecessary search dimensions, while negative evolution can potentially break the BBs and deteriorate the generalization performance.

III. SIMPLIFIED GENETIC OPERATORS OF GNP

The appearance of invalid evolution and negative evolution arises from the fact that uniform genetic operators do not consider the "transition by necessity" feature of GNP at all, resulting in meaningless and negative effect. To address these problems, we develop new genetic operators — *simplified genetic operators*. We name the proposal as GNP_simplified.

The fundamental concept of our proposal is to *strictly restrict the evolution behavior within the experienced/activated sub-graph of each directed graph*. In each generation, the untransited branches are retained as many as possible. The procedure of GNP_simplified is shown in Algorithm 2.

After the execution of each directed graph g, we record a set of its transited branches TB_g. For the entire population *Pop*, evolution is strictly restricted only to these sets of transited branches TB_g, $g \in Pop$, in each generation. The simplified genetic operators consist of two operators:

Algorithm 2: Procedure of GNP_Simplified

```
1  initialize population Pop;
2  repeat
3      for each individual g ∈ Pop do
4          execute g to obtain a set of transited branches TB_g;
5      preserve the best |Pop_e| individuals (elite selection);
       /* simplified crossover: step 6 to 10                    */
6      for index i = 0; i < |Pop_c|/2; i++ do
7          select two parent individuals P1 and P2 from Pop_c;
8          for each branch b ∈ TB_P1 ∪ TB_P2 do
9              if random_seed< p_c then
10                 swap the connection information of b between the parents;
       /* simplified mutation: step 11 to 15                    */
11     for index i = 0; i < |Pop_m|; i++ do
12         select a parent individual P from Pop_m;
13         for each branch b ∈ TB_P do
14             if random_seed< p_m then
15                 randomly set the connection information of b at [1, |S_node|];
16 until terminal conditions;
```

simplified crossover and *simplified mutation*. In simplified crossover (step 6–10), a branch is considered to be evolved only if it is transited in at least one of the parent individuals (step 8). In simplified mutation (step 11–15), only the transited branches can be evolved (step 13).

It is explicit that the proposal is simple and straightforward. Even though, we endorse the largest possibility to reduce the effect of invalid evolution. Figure 5 provides a graphical illustration of the search dimensions in simplified genetic operators. First, invalid evolution can be 100% avoided by simplified mutation, since a branch of mutation individual P can be evolved only if it is transited (belongs to TB_P, right part of Figure 5). Second, given two crossover individuals P1 and P2, a branch can be evolved only if it is either transited

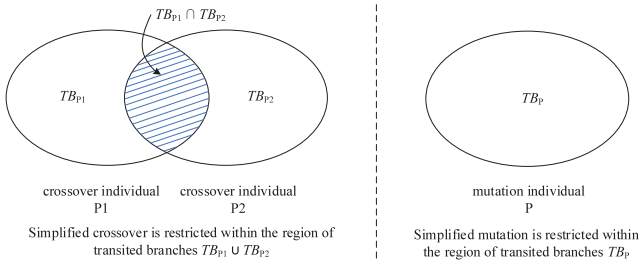

FIGURE 5. Search dimensions in simplified genetic operators.

in P1 or P2 (belongs to $TB_{P1} \cup TB_{P2}$, left part of Figure 5). If evolution behavior happens within $TB_{P1} \cap TB_{P2}$ region, no invalid evolution appears since the branches are transited in both crossover individuals. However, if crossover happens outside the intersection region (the blank region of left figure), one of the crossover branches will cause the invalid evolution. Therefore, simplified crossover can at least 50% reduce the invalid evolution (50% happens in the extreme case that $TB_{P1} \cap TB_{P2} = \emptyset$).

Summarizing the above discussion, we can calculate the *IER* of simplified genetic operators, i.e., *IER*(*simplified*), as follows.

For each simplified crossover action (with two crossover individuals P1 and P2), invalid evolution appears only if the branches satisfying $\in TB_{P1} \cup TB_{P2}$ and $\notin TB_{P1} \cap TB_{P2}$ are evolved. We denote the total number of these branches by $|IB|$, that,

$$IB = \{b | b \in TB_{P1} \cup TB_{P2} \quad \text{and} \quad b \notin TB_{P1} \cap TB_{P2}\}. \tag{12}$$

The theoretical number of untransited branches to be evolved is $0.5 \times |IB| \times p_c$.[5] Considering $|Pop_c|/2$ crossover actions in each generation, the total number is $0.5 \times \sum_{i=1}^{|Pop_c|/2} |IB_i| \times p_c$. Since there is no invalid evolution in simplified mutation, the total number of untransited branches to be evolved in each generation, i.e., *evol_UB*(*simplified*), can be calculated as follows.

$$evol\_UB(simplified) = 0.5 \times \sum_{i=1}^{|Pop_c|/2} |IB_i| \times p_c. \tag{13}$$

Simplified genetic operators restrict the evolution within the transited branches. Therefore, for each simplified crossover individual, the theoretical number of evolved branches is $|TB_g| \times p_c$, and that of each simplified mutation individual is $|TB_g| \times p_m$. Multiplied by the number of crossover/mutation individuals, we have the total number of evolved branches, i.e., *evol*(*simplified*), as follows.

$$evol(simplified) = \sum_{g \in Pop_c} |TB_g| \times p_c + \sum_{g \in Pop_m} |TB_g| \times p_m. \tag{14}$$

According to Definition 3, we can obtain *IER*(*simplified*) of each generation as follows.

$$IER(uniform) = \frac{evol\_UB(simplified)}{evol(simplified)} \times 100\%. \tag{15}$$

With respect to the negative evolution issue, it appears when the set of untransited branches which were previously transited (denoted as *NB* in Definition 5) is evolved for the first time. In simplified genetic operators, the size of *NB* dramatically decreases, since the antecedence of negative evolution is the evolution of untransited branches (invalid evolution) and it rarely happens in our proposal. Only simplified crossover will potentially cause the negative evolution, but with very small possibility. The calculation of *NER*(*simplified*) for simplified genetic operators is similar to that of *NER*(*uniform*) (denominator is slightly different), which is omitted here.

From the perspective of search dimensions, simplified genetic operators dramatically reduce the search dimensions comparing with uniform genetic operators.

Definition 6: Given a directed graph g of GNP, the set of search dimensions D in simplified genetic operators is defined by the total number of transited branches, i.e., $D = TB_g$.

Comparing with Definition 1, the number of search dimensions decreases from $\sum_{i \in S_{node}} L_i$ to $|TB_g|$, resulting in significant relaxation of problem complexity.

It is specifically notable that the proposal described in this paper is an enhanced version of an early work [18]. In the early work, the set of transited branches TB_g for each g is maintained throughout the entire generations. In each generation, TB_g is updated by adding new transited branches and removing the evolved ones (the branches become untransited once evolved). This will potentially increase the appearance of negative evolution to break the BBs without evaluation. Therefore, in the enhanced version, TB_g is generated per each generation, so that the negative evolution effect is reduced and the number of search dimensions also decreases accordingly.

As discussed previously, the fundamental motivation of this work arises from the theory of neuroscience and evolutionary Biology.

- *Introns* — First, evolution behavior is *only* carried out within the experienced/activated sub-graphs, whose qualities have been appropriately evaluated through the fitness calculation. The untransited sub-graphs can be considered as some kinds of unknown or potential skills, viewed as *introns* in neuroscience and evolutionary Biology. Because the qualities of *introns* are unknown, evolving them cannot result in meaningful improvement in the nature sense (invalid evolution), and might bring too much evolution pressure to deteriorate the evolution performance (negative evolution).
- *Atavism* — Second, simplified genetic operators are inspired by the *atavism* feature of evolutionary Biology. Atavism reflects that the phenotypically disappeared traits do not necessarily disappear from the DNA sequence. The traits often remain genotypically, but are inactive [26]. In each generation, the untransited branches refer to the phenotypically disappeared traits, which may (with high possibility) be experienced in the previous generations. Simplified genetic operators remain them as many as possible to keep the dormant genes to preserve the ability of atavism. Accordingly, the frequent breakage of BBs is avoided.

Conclusively, although only slight modifications of uniform genetic operators are made in our proposal, ''transition

[5] 0.5 denotes that only one branch in either P1 or P2 is an untransited branch causing invalid evolution.

TABLE 1. LIBRARY of maze problems and Tileworld.

Function	Description	#Branches
	Maze problems	
$J_1 \sim J_8$	Judge the eight adjacent cells	3 (floor, obstacle, goal)
$P_1 \sim P_8$	Move to the eight adjacent cells	1
	Tileworld	
$J_1 \sim J_4$	Judge forward, backward, left, right	5 (floor, obstacle, tile, hole, agent)
J_5	Judge the direction of the nearest tile	5 (four directions, cannot find)
J_6	Judge the direction of the nearest hole	
J_7	Judge the direction of the nearest hole from the nearest tile	
J_8	Judge the direction of the second nearest tile	
$P_1 \sim P_4$	Move forward, turn left, turn right, stay	1

by necessity" feature of GNP is explicitly considered for evolution. The systematic and theoretical analysis provides necessary evidences to demonstrate the superiority of simplified genetic operators: 1) The invalid evolution and negative evolution problems are significantly reduced; 2) The number of search dimensions is significantly reduced; 3) The BBs are retained as many as possible.

IV. EXPERIMENTAL STUDY

The experiments are carried out in two benchmark testbeds of the agent control problems, i.e., maze problems [19] and Tileworld [20].

A. BENCHMARK TESTBEDS

1) MAZE PROBLEMS

Maze problems consist of an artificial agent and a grid world. The agent has eight sensors to recognize the surrounding cells, and makes eight possible actions for the adjacent movement. There are three kinds of cells — obstacle, floor or goal cell. The target of this problem is to control the agent to find the goal cell with the fewest number of steps wherever it is initially located. In other words, it is a minimization problem with fitness defined as *steps to goal*. As shown in Figure 6, two well-known maps are studied, named `Woods1` and `Maze5` [19], where the optimal steps are 1.7 and 4.6 steps, respectively.

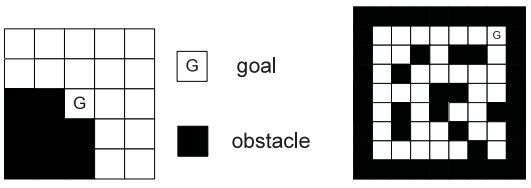

FIGURE 6. Tested maze maps.

a: FITNESS CALCULATION

When performing each individual run, twenty instances are tested. In each instance, the agent is randomly assigned in the world, and the instance is finished once the agent finds the goal cell or reaches the predefined maximal steps, i.e., 2000. The fitness of each individual is calculated by the average steps to goal of twenty instances.

b: DIRECTED GRAPH DESIGN

For maze problems, we design eight judgment functions to judge the agent's eight adjacent cells. The judging results could be obstacles, floors or a goal, so resulting in three branches. Eight processing functions are performed to move the agent to its adjacent cells. The LIBRARY of maze problems are defined in Table 1.

2) TILEWORLD

Tileworld benchmark testbed represents a more complex system than maze problems. It is constructed by a grid world with agents, obstacles, floors, holes and tiles (Figure 7). There are plural agents, where each agent is moved simultaneously. The target of this problem is to control the agents to work in a cooperative way to push tiles into holes within predefined maximal steps. As presented in Figure 8, ten Tileworld maps are tested. In these cases, the number of agents are three. The worlds are designed with the same positions of agents, obstacles and holes, where the tiles are initially placed in different cells. Ten worlds are learned simultaneously when evolving each individual.

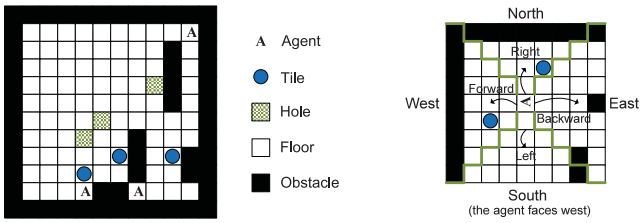

FIGURE 7. Tileworld environment and the agent's directional recognition ability.

a: FITNESS CALCULATION

The fitness of an individual per each map is calculated by:

$$f = 100 \times DT + 3 \times (ST - ST_{\text{used}}) + 20 \times \sum\nolimits_{t \in Tile} (D_t - d_t). \quad (16)$$

DT refers to the number of tiles dropped into holes. ST is the predefined constrained steps. ST_{used} is the number of used steps. $Tile$ is the set of tiles. D_t represents the initial distance from tile t to its nearest hole, and d_t denotes the final distance from t to its nearest hole after ST steps.

The three terms of function f correspond to three targets of Tileworld: 1) We should drop tiles into holes as many as possible, denoted by $100 \times DT$; 2) We should drop tiles into holes using as few steps as possible, denoted by $3 \times (ST - ST_{\text{used}})$; 3) If we cannot drop all tiles within ST steps, we should push the remaining ones to holes as near as possible, represented by $20 \times \sum_{t \in Tile}(D_t - d_t)$. Since ten Tileworld maps are tested simultaneously, and so the final fitness value is counted by $\sum_{world=1}^{10} f(world)$.

b: DIRECTED GRAPH DESIGN

For Tileworld, we design eight judgment functions to judge the agent's surrounding environment, which are listed in Table 1. The agents are designed with four abilities of directional recognition (forward, backward, left, and right), where the definitions are shown in the example of Figure 7 (right). With judgment functions $J_1 \sim J_4$, the agent can perceive the surrounding cells of four directions. Judgment functions $J_5 \sim J_8$ are designed to judge the directional information of the nearest or second nearest tiles/holes. Each agent has four action abilities, i.e., move forward, turn left, turn right and stay.

3) COMPARISON METRICS

We specifically note that all the experimental results are the average of 30 independent trials in order to eliminate the random bias for fair comparison. Two comparison metrics are performed in the paper.

a: METRIC 1—FITNESS RESULTS

The first and primary metric to compare different algorithms is the final *fitness result* after user-defined terminal criteria, i.e., reaching the maximal fitness evaluations, which is commonly used in the vast majority of evolutionary computation studies.

b: METRIC 2—REQUIRED FITNESS EVALUATIONS

In order to testify the search speed, the second comparison metric is the *required fitness evaluations* (RFEs). The RFEs are calculated in the following way: for each successful trial,[6] the used FEs are counted; for each failed trial, the maximal FEs are counted; Afterwards, we average the RFEs of 30 independent trials for comparison.

Both metrics are counted to conduct a comprehensive validation of the proposal to compare with the state-of-the-art algorithms. In addition, two-tailed, paired *t-test* [12], [27] is performed to demonstrate the statistical significance.

B. EXPERIMENTAL STUDIES ON MAZE PROBLEMS

1) STUDIED ALGORITHMS AND PARAMETER CONFIGURATION

To demonstrate the superiority of the proposed algorithm, four state-of-the-art algorithms are selected as follows.

Extended classifier system (XCS) [28]: XCS is the most well-known evolutionary classifier system (learning classifier system, or LCS) [29] which learns and evolves a population of classifiers for decision making. Its classifiers are represented as ''IF-THEN'' type decision rules. The rules are encoded by ternary alphabet, i.e., {0, 1, #}. Symbol # means ''don't care'' which matches with 0 and 1. From this perspective, XCS formulates its solutions in a similar way as GNP, which ensures strong generalization ability through decision rules to cover plural perceptions. XCS has performed outstandingly to solve the maze problems [30]. The parameter settings of XCS are carried out based on the suggestion [31] to perform the best results, which are presented in Table 2.

Genetic programming (GP) [5]: GP is selected as the representative of conventional EAs. Different from GNP, GP applies trees as its individual representation. The non-leaf nodes of GP trees work in a way as GNP's judgment nodes, while the leaf nodes are composed of processing functions. We encode the complete three-ary trees for GP programs since all the judgment functions have three arguments. In order to obtain the best GP performance, the maximal tree-depth is defined to five (tested and selected from tree-depth three, four, five and six). In this case, GP tree consists of 361 nodes. FULL initialization, one-point crossover and point mutation [32] are used.[7]

Sarsa learning (Sarsa) [33]: Sarsa is selected as the representative of reinforcement learning (RL) technique. We define the Sarsa states as the total possible observations of each agent in its eight directional cells. Accordingly, the total number of states is $3^8 = 6561$. The actions are defined as the processing functions of Table 1, which are eight. ε-greedy policy is used for the selection of actions.

GNP_uniform and GNP_simplified: standard GNP with uniform genetic operators (GNP_uniform for short) is selected as the baseline to compare with the proposed GNP_simplified. The overall parameters of GNP variants include: population size (elite size, crossover size, and mutation size); the directed graph G (node size $|S_{\text{node}}|$, and node size for each judgment/processing function); crossover rate and mutation rate.

Hand-tuning strategy is applied to determine the parameters of each compared algorithm, which are listed in Table 2. That is, a large number of parameter settings are tested, and the ones achieving the best performance are used to ensure the fair comparison.

All the experiments are performed on a PC with Intel Core i7 running at 3.40GHz with 8 GB RAM. The complier is Eclipse Neon (ver. 4.6.3) of Windows 10. Several open source codes are used to implement the compared algorithms, including XCSJava 1.0 [31] for XCS and ECJ library [34] for GP.

[6] A successful trial is defined by: (1) Maze problems: for twenty instances with randomly initial positions, the agent can find the goal cell using the optimal steps; (2) Tileworld: all tiles can be dropped into the holes within ST steps.

[7] We tested different initialization, crossover and mutation methods, and used the best options for the studied testbeds.

TABLE 2. Parameter settings of the compared algorithms.

Algorithm	Detailed parameters
	Maze problems
	`Woods1` map
XCS	maximal #classifiers 800; learning rate 0.2; discount factor 0.7; error threshold 10; $P_{\#} = 0.6$; crossover rate 0.8; mutation rate 0.04; GA threshold 25.
GP	population size $\|Pop\| = 40$; elite size $\|Pop_e\| = 1$; crossover size $\|Pop_c\| = 20$; mutation size $\|Pop_m\| = 19$; crossover rate $p_c = 0.9$; mutation rate $p_m = 0.02$; $\|S_{\text{node}}\| = 361$ (maximal tree depth 5).
Sarsa	learning rate 0.1; discount factor 0.9; ε-greedy policy $\varepsilon = 0.1$.
GNP_uniform (GNP_simplified)	population size $\|Pop\| = 40$; elite size $\|Pop_e\| = 1$; crossover size $\|Pop_c\| = 20$; mutation size $\|Pop_m\| = 19$; crossover rate $p_c = 0.2$; mutation rate $p_m = 0.03$; $\|S_{\text{node}}\| = 80$ (#node per each judgment/processing function= 5).
	`Maze5` map
XCS	maximal #classifiers 3000; learning rate 0.2; discount factor 0.7; error threshold 5; $P_{\#} = 0.3$; crossover rate 0.8; mutation rate 0.01; GA threshold 25.
GP	population size $\|Pop\| = 40$; elite size $\|Pop_e\| = 1$; crossover size $\|Pop_c\| = 20$; mutation size $\|Pop_m\| = 19$; crossover rate $p_c = 0.9$; mutation rate $p_m = 0.03$; $\|S_{\text{node}}\| = 361$ (maximal tree depth 5).
Sarsa	learning rate 0.2; discount factor 0.9; ε-greedy policy $\varepsilon = 0.15$.
GNP_uniform (GNP_simplified)	population size $\|Pop\| = 40$; elite size $\|Pop_e\| = 1$; crossover size $\|Pop_c\| = 20$; mutation size $\|Pop_m\| = 19$; crossover rate $p_c = 0.3$; mutation rate $p_m = 0.04$; $\|S_{\text{node}}\| = 80$ (#node per each judgment/processing function= 5).
	Tileworld
GP	population size $\|Pop\| = 300$; elite size $\|Pop_e\| = 1$; crossover size $\|Pop_c\| = 120$; mutation size $\|Pop_m\| = 179$; crossover rate $p_c = 0.9$; mutation rate $p_m = 0.01$; $\|S_{\text{node}}\| = 781$ (maximal tree depth 4).
Sarsa	learning rate 0.2; discount factor 0.9; ε-greedy policy $\varepsilon = 0.1$.
GNP_uniform (GNP_simplified)	population size $\|Pop\| = 300$; elite size $\|Pop_e\| = 1$; crossover size $\|Pop_c\| = 120$; mutation size $\|Pop_m\| = 179$; crossover rate $p_c = 0.1$; mutation rate $p_m = 0.02$; $\|S_{\text{node}}\| = 60$ (#node per each judgment/processing function= 5).

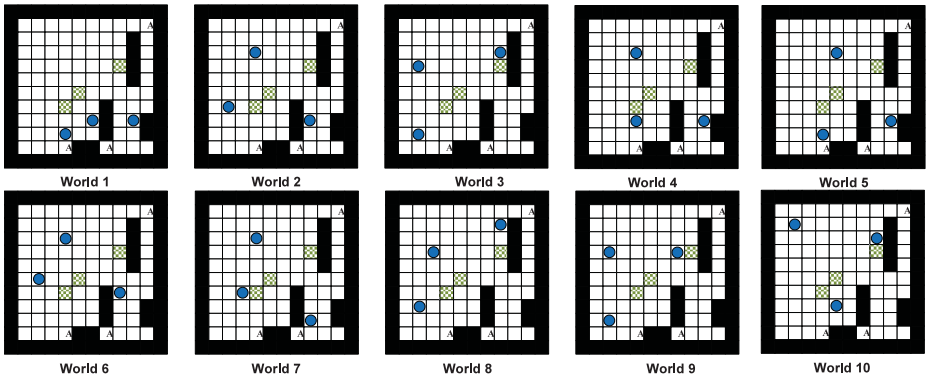

FIGURE 8. Tileworld maps.

2) RESULTS AND ANALYSIS

a: FITNESS RESULTS

In maze problems, the maximal fitness evaluations are set to 5,000 to terminate each experimental trial. The fitness curves in `Woods1` and `Maze5` are shown in Figure 9 and 10, where the detailed fitness results are listed in Table 3. In `Woods1`, XCS, GP, Sarsa, GNP_uniform and GNP_simplified are all capable of reaching the goal cell with the optimal steps (1.7 steps). In `Maze5`, GP and GNP_uniform fail to achieve the optimal steps, while XCS, Sarsa and GNP_simplified converge to the optimum (4.6 steps).

b: REQUIRED FITNESS EVALUATIONS

The RFEs are further counted to reflect the search speed. The results are listed in Table 3. In `Woods1`, both GNP_uniform and GNP_simplified achieve faster search speed than the other compared algorithms. In `Maze5`, GP and GNP_uniform cannot solve the task, which are surpassed by XCS, Sarsa and GNP_simplified. Throughout the two maze maps, GNP_simplified can successfully find the goal cell with the fewest RFEs.

c: STATISTICAL TEST

The t-test results (with 95% confidence level) are shown in the "Sig." columns of Table 3. Except GP and GNP_uniform for `Maze5` map, the "Sig." results of fitness show that all the compared algorithms identically reach the optimal steps with no statistical difference. The "Sig." results of RFEs demonstrate the statistically significant difference between GNP_simplified and the compared algorithms.

TABLE 3. Detailed results in maze problems.

	Woods1				Maze5			
	Fitness	Sig.	RFEs	Sig.	Fitness	Sig.	RFEs	Sig.
XCS	**1.7±0.1**	7.21e-1	548±118	**1.77e-12**	**4.6±0.1**	8.58e-1	1151±278	**1.95e-3**
GP	**1.7±0.1**	7.87e-1	336±103	**7.63e-5**	5.2±0.1	**7.65e-3**	5000±0	**8.12e-42**
Sarsa	**1.7±0.1**	7.21e-1	663±125	**9.35e-15**	**4.6±0.1**	7.62e-1	1553±332	**6.31e-6**
GNP_uniform	**1.7±0.1**	2.30e-1	305±91	**3.36e-3**	4.9±0.1	**3.35e-2**	5000±0	**8.12e-42**
GNP_simplified	**1.7±0.1**	—	**247±78**	—	**4.6±0.1**	—	**959±168**	—

[1] In "Fitness" and "RFEs" columns, the **bold** values denote the best results.
[2] "Sig." (significance) columns list the p-values of statistical t-test (GNP_simplified as the base algorithm), and the **bold** values denote the statistically significant difference (smaller than 0.05).

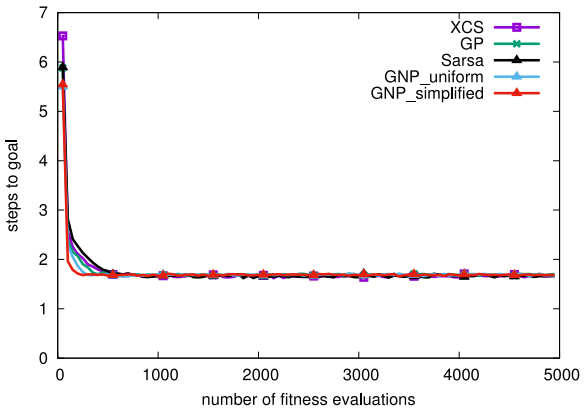

FIGURE 9. Performance in Woods1 map.

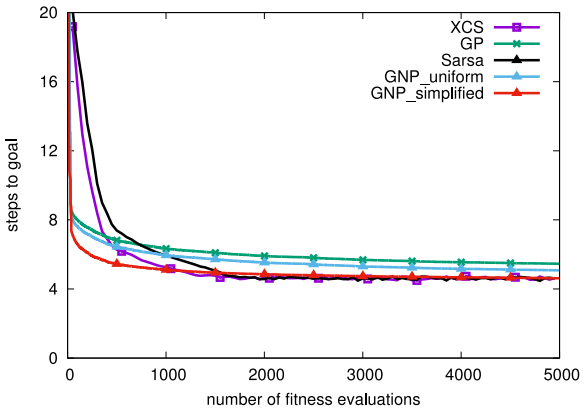

FIGURE 10. Performance in Maze5 map.

d: PERFORMANCE ANALYSIS

In the simple Woods1, GNP family achieves better performance than the compared state-of-the-art algorithms in the early stages of evolution (faster search speed). After a small number of fitness evaluations (less than 550 FEs), all the compared algorithms quickly locate the optimal fitness results with stable and smooth convergence. Maze5 is a much more difficult problem since much less generalizations are possible [30]. Different from Woods1 whose left and right edges are connected, Maze5 places the four edges with obstacles, formulating much more complicated scenarios. In this map, GNP family also performs faster convergence than the compared algorithms in the early fitness evaluations. However, XCS and Sarsa gradually achieve better performance than GP and GNP_uniform, which eventually converge to the optimal steps. GP and GNP_uniform can only reach the near optimal steps.

Overall, the proposed GNP_simplified statistically achieves the best performance in terms of both final fitness results and RFEs comparing with the state-of-the-art XCS, GP, Sarsa and GNP_uniform. From this perspective, it is empirically proved that the proposed GNP_simplified overcomes the drawbacks of traditional genetic operators (GNP_uniform) to achieve the best performance.

C. EXPERIMENTAL STUDIES ON TILEWORLD

1) STUDIED ALGORITHMS AND PARAMETER CONFIGURATION

To further demonstrate the effectiveness of the proposed algorithm, a more complicated system Tileworld benchmark testbed is performed, and the following state-of-the-art algorithms are selected for comparison.

GP: The complete five-ary trees are encoded for GP programs since all the judgment functions have five arguments. The maximal tree-depth is defined to four (tested and selected from tree-depth three, four and five) to perform the best results of GP. In this case, GP tree consists of 781 nodes. FULL initialization, one-point crossover and point mutation [32] are used.

Sarsa [33]: We define the Sarsa states as the total possible observations of each agent in its four directional cells. Accordingly, the total number of states is $5^4 = 625$. The actions are defined as the processing functions of Table 1, which are four. Overall, there are $625 \times 4 = 2500$ sate-action pairs to be learned in Sarsa. ε-greedy policy is used for the selection of actions.

GNP_uniform is performed as the baseline to compare with the proposed GNP_simplified. All the detailed parameters of the compared algorithms are set by hand-tuning strategy, presented in Table 2. As the experiments on maze problems, a large number of parameter settings are tested, where the ones with the best results are used for the fair comparison.

2) RESULTS AND ANALYSIS

a: PROBLEM DIFFICULTY AND SCALABILITY

Given a specific Tileworld map, our target is to drop tiles into holes within a user-defined constrained steps, that is,

TABLE 4. Detailed fitness results in Tileworld under different *ST* values.

	Fitness	Sig.	*DT*	Fitness	Sig.	*DT*
	$ST = 40$			$ST = 60$		
GP	3139.9±482.3	**3.36e-6**	16.3±2.5	3356.3±679.2	**2.28e-9**	17.6±3.5
Sarsa	2971.7±492.5	**2.53e-7**	15.1±2.5	3719.2±1106.2	**8.53e-4**	18.9±5.1
GNP_uniform	3532.1±540.4	**8.75e-3**	18.3±2.7	4171.4±692.0	**2.12e-2**	23.0±3.8
GNP_simplified	**3950.3±563.2**	—	**21.3±2.7**	**4595.3±602.3**	—	**25.3±2.8**
	$ST = 80$			$ST = 100$		
GP	3802.5±789.8	**3.59e-5**	19.3±4.1	3830.0±1080.9	**7.62e-8**	19.5±4.8
Sarsa	4031.1±802.2	**8.36e-4**	21.5±4.2	4463.1±809.3	**1.79e-4**	24.7±4.3
GNP_uniform	4337.0±773.4	**1.02e-2**	24.3±4.0	4974.3±556.1	**2.12e-2**	27.5±2.7
GNP_simplified	**4873.2±953.2**	—	**27.0±1.5**	**5412.3±897.5**	—	**29.2±1.3**

[1] "*DT*" columns present the number of dropped tiles by each algorithm.
[2] "Sig." columns present the statistical tests of the fitness results.

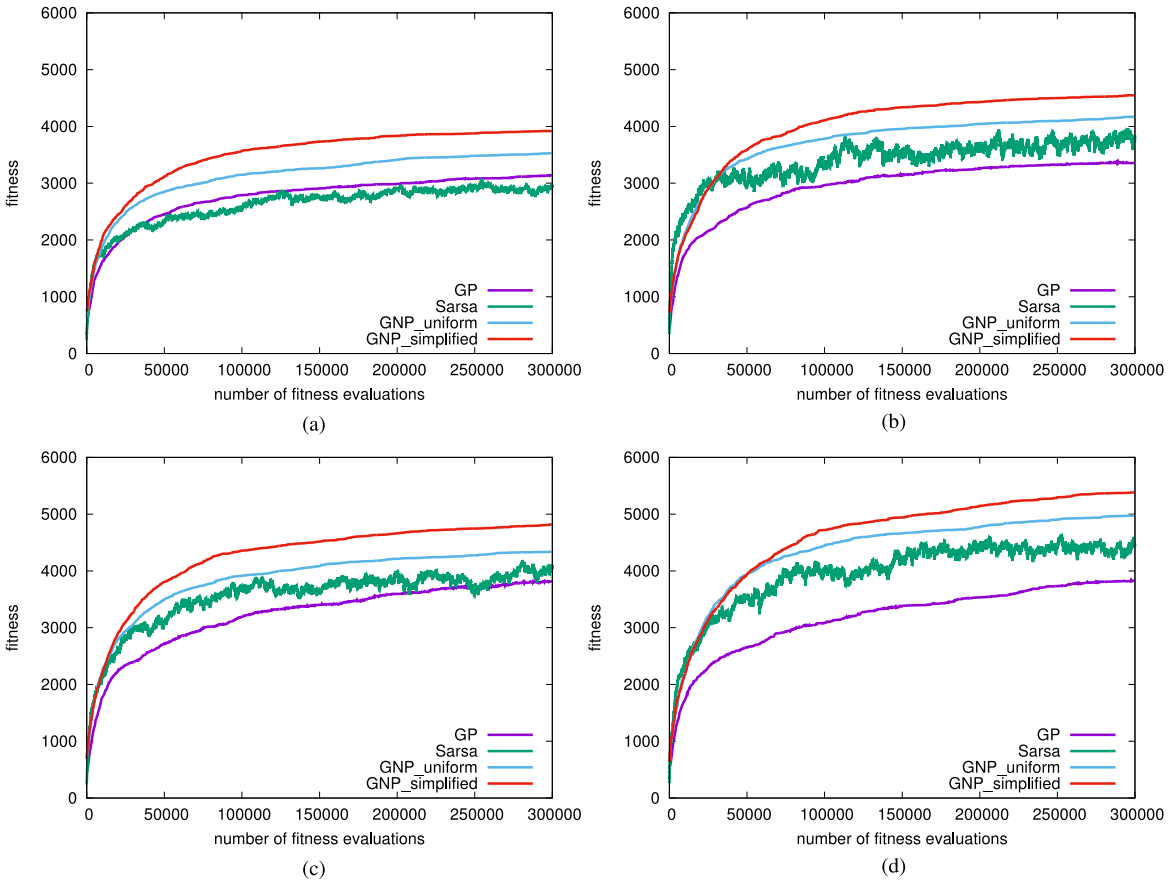

FIGURE 11. Performance in Tileworld under different *ST* values. (a) $ST = 40$. (b) $ST = 60$. (c) $ST = 80$. (d) $ST = 100$.

parameter *ST* in the fitness Equation (16). Therefore, *ST* plays the essential role to control the problem difficulty of Tileworld. With the decrease of *ST* values, we should use smaller steps to solve the tasks. In other words, smaller *ST* values indicate more difficult problems. Different *ST* values allow us to validate the algorithm scalability under different problem difficulty. For this purpose, we set four *ST* values to design the experiments, that, $ST = \{40, 60, 80, 100\}$.

b: FITNESS RESULTS

In Tileworld, the maximal fitness evaluations are set to 300,000 to terminate each experimental trial. The fitness curves under different *ST* values are shown in Figure 11, where the detailed fitness results are listed in Table 4. Throughout the studied four cases, GNP family surpasses the state-of-the-art GP and Sarsa, where the proposed GNP_simplified achieves the best fitness results.

TABLE 5. Detailed RFEs in Tileworld under different *ST* values.

	RFEs	Sig.	RFEs	Sig.
	$ST = 40$		$ST = 60$	
GP	300,000±0	**8.92e-3**	300,000±0	**5.82e-7**
Sarsa	300,000±0	**8.92e-3**	289,492±31,699	**2.58e-6**
GNP_uniform	296,859±12,097	**2.14e-2**	275,361±57,308	**1.62e-2**
GNP_simplified	**286,200±26,961**	—	**232,707±57,871**	—
	$ST = 80$		$ST = 100$	
GP	288,159±27,073	**7.05e-8**	287,785±25,478	**4.51e-14**
Sarsa	278,947±46,808	**2.57e-6**	253,210±67,210	**2.24e-8**
GNP_uniform	260,747±65,960	**5.53e-4**	199,662±78,862	**1.12e-2**
GNP_simplified	**205,720±58,527**	—	**156,720±53,885**	—

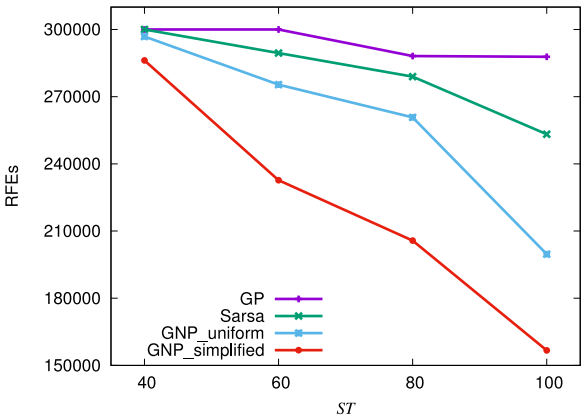

FIGURE 12. Graphical illustration of RFEs in Tileworld under different *ST* values.

c: REQUIRED FITNESS EVALUATIONS

The results of RFEs are shown in Table 5. Meantime, we plot the graphical illustration of RFEs under different *ST* values in Figure 12. In the difficult problems of $ST = 40$ or 60, GP and Sarsa fail to solve the tasks, and they can only solve the simpler problems but with large RFEs. Overall, the proposed algorithm requires the fewest RFEs.

d: STATISTICAL TEST

The t-test results of fitness and RFEs are shown in the "Sig." columns of Table 4 and 5, respectively. The "Sig." results show that there are statistically significant differences between GNP_simplified and the other algorithms in terms of both fitness results and RFEs.

e: PERFORMANCE ANALYSIS

In order to evaluate the success/failure of problem solving, we further count the the number of dropped tiles for comparison, explicitly reflecting whether all tiles have been dropped into holes within *ST* steps. The results are reported in the "*DT*" columns of Table 4. When setting *ST* to small values (i.e., 40), it is difficult to drop tiles within such small constrained steps. All methods cannot 100% solve the task throughout 30 independent trials. Particularly, GP and Sarsa cannot achieve a single successful trial, resulting smallest fitness results and 300,000 RFEs (maximal FEs), while GNP family succeeds in some trials. When increasing *ST* values, the problem difficulty is relaxed that we have more steps to push tiles. The "*DT*" results show that GNP_simplified can drop more tiles into holes than the other algorithms, which indicates that our proposal has more chance to successfully solve the Tileworld tasks.

Considering all the presented results as a whole, the experiments explicitly demonstrate the statistical superiority of GNP_simplified over the other state-of-the-art algorithms throughout different problem difficulties.

D. GENERALIZATION PERFORMANCE

In addition to compare the fitness results and RFEs in the given (training) environments, we further testify the trained best solutions of each algorithm under inexperienced (testing) environments to demonstrate the generalization performance.

In maze problems, since the agent's initial location is randomly placed, the fitness results (average steps) presented in Table 3 are capable of reflecting not only the evolution efficiency of the studied algorithm, but also its generalization performance [8] in certain degrees. In Table 3, the experimental results demonstrate that the proposed GNP_simplified reaches the best generalization performance with the fewest RFEs comparing with the other compared state-of-the-art algorithms.

In Tileworld, the training experiments are carried out under static environments. Therefore, we further design additional testing experiments to demonstrate the generalization performance. Two scenarios of testing environments are designed based on the Tileworld maps of Figure 8.

- T1 (simple case): Randomly set the initial positions of tiles, where the other objects are remained as the training maps.
- T2 (complicated case): Randomly set the initial positions of tiles, holes and obstacles.

The testing experiments of each scenario are conducted as follows. First, 1000 testing environments are randomly generated. Second, each final solution obtained in the training environment is applied to obtain an average testing result (of 1000 testing environments). For each compared algorithm, the final testing result is the average of its 30 final training solutions obtained in the second step. Finally, we can compare the generalization performance of each algorithm via the final testing results.

For two different testing scenarios, the compared results are shown in Table 6 and 7. The testing results of both simple and complicated testing scenarios clearly demonstrate that the proposed GNP_simplified achieve higher generalization performance than the other compared algorithms.

Overall, the testing experiments constitutes our argument that the proposed simplified genetic operators benefit to significantly reduce the effect of negative evolution, so that the BBs can be retained as many as possible to ensure the generalization performance.

[8]The solution can adapt to the tested maze map, wherever the agent is initially placed.

TABLE 6. Detailed testing results of T1 scenario in Tileworld.

	Fitness	Sig.	DT	Fitness	Sig.	DT
	$ST = 40$			$ST = 60$		
GP	1196.5±568.3	**7.31e-7**	6.8±2.7	1369.6±752.0	**5.86e-9**	7.6±3.3
Sarsa	1074.7±431.5	**1.28e-9**	6.3±3.1	1879.2±851.5	**6.47e-7**	8.9±3.8
GNP_uniform	1532.0±668.3	**3.35e-4**	8.0±2.1	2276.3±933.0	**5.33e-3**	11.7±3.9
GNP_simplified	**2037.3±776.5**	—	**10.3±2.3**	**2475.3±832.3**	—	**13.3±3.2**
	$ST = 80$			$ST = 100$		
GP	1950.5±765.5	**8.52e-8**	9.5±4.3	1957.0±785.9	**7.43e-9**	9.6±4.5
Sarsa	2152.1±772.5	**3.98e-7**	11.2±4.6	2413.1±706.5	**5.32e-6**	12.9±4.3
GNP_uniform	2392.0±668.3	**3.33e-4**	12.6±3.7	2812.3±621.1	**7.85e-3**	14.5±3.7
GNP_simplified	**2731.2±758.2**	—	**14.1±3.2**	**3073.3±798.5**	—	**15.9±2.9**

TABLE 7. Detailed testing results of T2 scenario in Tileworld.

	Fitness	Sig.	DT	Fitness	Sig.	DT
	$ST = 40$			$ST = 60$		
GP	182.2±885.4	**3.36e-6**	1.4±5.5	213.3±843.2	**2.35e-9**	1.5±4.5
Sarsa	168.3±695.4	**2.53e-7**	1.2±6.5	321.2±665.2	**8.31e-7**	2.3±4.1
GNP_uniform	271.1±745.8	**8.75e-3**	2.0±6.7	547.3±875.0	**5.21e-4**	3.6±5.8
GNP_simplified	**437.3±761.8**	—	**3.0±5.7**	**787.3±782.3**	—	**5.6±4.8**
	$ST = 80$			$ST = 100$		
GP	362.5±686.6	**1.55e-8**	2.6±4.3	367.0±680.2	**1.21e-10**	2.6±4.5
Sarsa	492.1±847.2	**2.31e-7**	3.3±4.6	831.1±599.3	**7.56e-6**	5.9±4.1
GNP_uniform	762.0±753.3	**1.33e-3**	5.5±4.3	977.3±455.1	**8.53e-3**	6.6±2.9
GNP_simplified	**933.2±552.2**	—	**6.3±3.5**	**1201.3±493.5**	—	**7.0±1.9**

E. DETAILED ANALYSIS OF GNP_SIMPLIFIED

The fundamental basis of the proposed algorithm relies on the unique feature of GNP's directed graph structure: "transition by necessity", targeting to solve the invalid evolution and negative evolution problems caused by the traditional uniform genetic operators. The following experiments address the effects of the proposed simplified genetic operators through the discussion of the evolution behavior and computation time. The experiments are carried out in Tileworld (with four ST values studied in section IV-C).

1) ANALYSIS OF EVOLUTION BEHAVIOR

In order to demonstrate the positive effects of the simplified genetic operators, its evolution behavior is analyzed in comparison with the uniform genetic operators.

As illustrated in Example 2, each directed graph of GNP only transits some branches (activate a specific sub-graph) to adapt to the perceived environments. When some untransited branches are evolved, invalid evolution appears (detailed analysis in section II-E.1). This problem is measured by a metric — invalid evolution rate (*IER*), defined by the rate that the untransited branches are evolved over the total evolved branches (Definition 3). The first evidence to prove the advantage of GNP_simplified over GNP_uniform arises from the fact that simplified genetic operators would result in much smaller *IER* values than the traditional uniform genetic operators. The comparison of *IER* values is plotted in Figure 13. Overall, GNP_uniform achieves 40% *IER* value, which means that 40% evolution actions are invalid, resulting meaningless evolution. Meantime, GNP_simplified endorses the largest possibility to reduce the invalid evolution, where

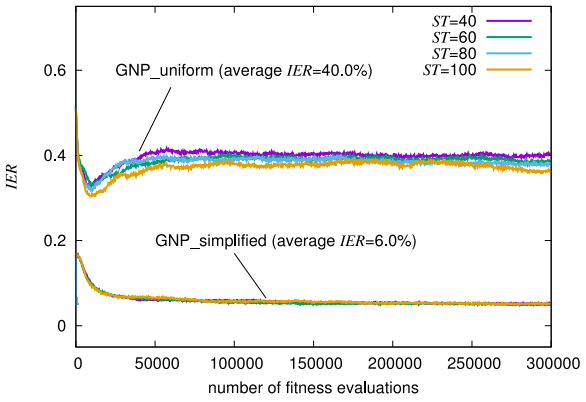

FIGURE 13. Comparison of *IER* values in Tileworld under different *ST* values.

only 6% evolution actions are invalid. The reason is straightforward as discussed in section III, that simplified mutation can completely avoid the invalid evolution, and simplified crossover can at least 50% reduce the problem.

In addition to the invalid evolution issue, another advantage of GNP_simplified refers to the negative evolution problem. As discussed in section II-E.2, when the untransited branches which were activated in the previous generations are evolved, the frequent breakage of BBs would appears to result in negative evolution. To quantitatively analyze this issue, we measure negative evolution rate (*NER*, Definition 5) of each algorithm for comparison. The comparison of *NER* values is plotted in Figure 14. Overall, 20.6% evolution actions of GNP_uniform are negative (average $NER = 20.6\%$). Meantime, GNP_simplified significantly reduces this value to only 3.1%.

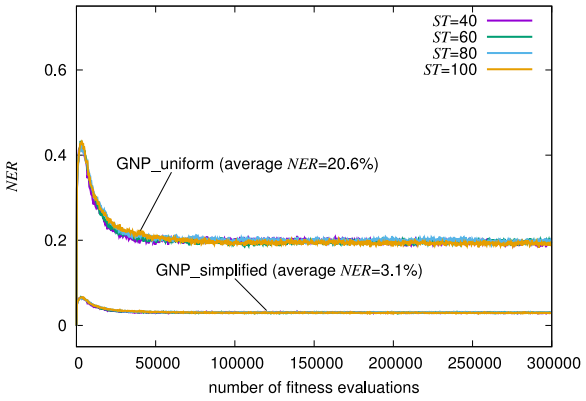

FIGURE 14. Comparison of *NER* values in Tileworld under different *ST* values.

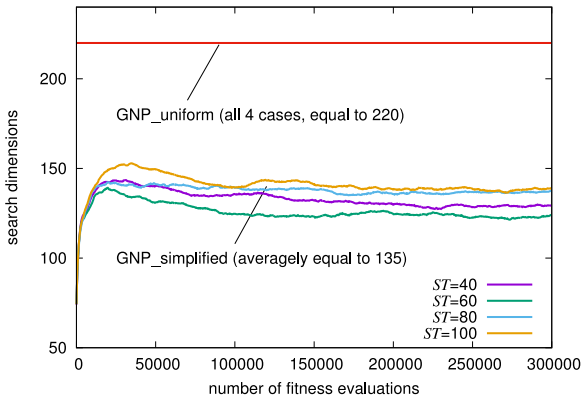

FIGURE 15. Change of search dimensions in Tileworld under different *ST* values.

Furthermore, GNP_simplified tends to reduce the search dimensions (defined by the evolvable branches) comparing with GNP_uniform (Definition 1 and 6). GNP_uniform treats all branches of the directed graph as the search dimensions, regardless of whether they are transited or not. On the other hand, GNP_simplified only evolves the activated branches of the directed graph, which are adaptively changed generation by generation. Accordingly, GNP_simplified is capable of significantly reducing the search dimensions, so that the evolution power is biased towards recombining the experienced sub-graphs, allowing higher evolution ability and efficiency. As shown in the experiments of Tileworld (Figure 15), the proposed algorithm reduce the search dimensions from 220 to 135 (38.6% reduction).

2) ANALYSIS OF COMPUTATION TIME

Finally, we compare the computation time between GNP_uniform and GNP_simplified in Table 8. The experiments are performed in Tileworld testbed on a PC with Intel Core i7 running at 3.40GHz with 8 GB RAM. The complier is Visual Studio 2012 of Windows 10.

GNP_simplified enforces the evolution actions within the activated, experienced branches, resulting in the reduction of search dimensions. This explicitly brings the advantage of requiring less computation time than the traditional GNP_uniform. The results show that GNP_simplified achieves 31.1% acceleration rate comparing with GNP_uniform.

TABLE 8. Comparison of computation time in Tileworld.

	GNP_uniform	GNP_simplified
Computation time (unit: sec.)	249.0	171.5
Acceleration rate	—	31.1%

[1] Acceleration rate is calculated by $(249.0 - 171.5)/249.0 = 31.1\%$.

F. DISCUSSIONS

Considering the experimental studies in maze problems and Tileworld as a whole, the proposed GNP_simplified is proved to achieve better evolution results (higher fitness results) and faster search speed (fewer RFEs) comparing with traditional GNP_uniform and the state-of-the-art algorithms of LCS, EAs, and RL. The detailed analysis of evolution behavior demonstrates that

- The invalid evolution and negative evolution problems are widespread when evolving the directed graphs of GNP.
- Based on the quantitative studies, it is empirically proved that uniform genetic operators result in serious invalid evolution and negative evolution problems ($IER = 40.0\%$ and $NER = 20.6\%$ in Tileworld).
- By explicitly taking the "transition by necessity" feature into account, simplified genetic operators achieve the significant reduction of invalid/negative evolution ($IER = 6.0\%$ and $NER = 3.1\%$ in Tileworld).
- With the proposal of simplified genetic operators, the search dimensions of GNP are significantly reduced, resulting the simplification of evolution and reduction of computation time.
- All the experimental results provide empirical evidence to support the theoretical analysis presented in this paper.

V. CONCLUSIONS

GNP distinguishes itself from the vast majority of EAs by its unique directed graph structure. However, the systematic and theoretical studies with respect to this issue are relatively unexplored in the literature. In this paper, we present a detailed study of GNP from both of theoretical and empirical viewpoints: 1) We explicitly reveal the "transition by necessity" feature of GNP's directed graph; 2) We proved that when ignoring this feature, uniform genetic operators tend to cause serious problems of invalid evolution and negative evolution. Following our discovery, simplified genetic operators are developed, which only evolve the activated/transited branches of GNP. We theoretically prove that our proposal is capable of reducing the effect of invalid/negative evolution. Meantime, the empirical studies in two benchmark testbeds — maze problems and Tileworld — demonstrate the effectiveness and superiority of our proposal over the state-of-the-art algorithms.

The primary target of this work is to develop specific genetic operators of GNP. However, it is expected that simplified genetic operators can be considered as a general and suitable strategy for GNP evolution due to its clear inspiration and simple implementation. Therefore, the future work will be to explore this study to the wide range of GNP variants and their real-world applications.

REFERENCES

[1] J. H. Holand, *Adaptation in Natural and Artificial Systems*. Ann-Arbor, MI, USA: Univ. Michigan Press, 1975.
[2] N. Hansen, "The CMA evolution strategy: A comparing review," in *Towards a New Evolutionary Computation* (Studies in Fuzziness and Soft Computing), vol. 192. Berlin, Germany: Springer, 2006, pp. 75–102.
[3] J. J. Liang, A. K. Qin, P. N. Suganthan, and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," *IEEE Trans. Evol. Comput.*, vol. 10, no. 3, pp. 281–295, Jun. 2006.
[4] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Trans. Evol. Comput.*, vol. 15, no. 1, pp. 4–31, Feb. 2011.
[5] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.
[6] A. Blanco, M. Delgado, and M. C. Pegalajar, "A real-coded genetic algorithm for training recurrent neural networks," *Neural Netw.*, vol. 14, no. 1, pp. 93–105, 2001.
[7] D. H. Wolper and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.
[8] A. E. Teller and M. Veloso, "PADO: Learning tree structured algorithms for orchestration into an object recognition system," Dept. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-95-101, 1995.
[9] K. Hirasawa, M. Okubo, H. Katagiri, J. Hu, and J. Murata, "Comparison between genetic network programming (GNP) and genetic programming (GP)," in *Proc. Congr. Evol. Comput.*, May 2001, pp. 1276–1282.
[10] J. F. Miller, "Cartesian genetic programming," *Natural Comput.*, vol. 10, no. 2, pp. 17–34, 2011.
[11] S. Mabu, K. Hirasawa, and J. Hu, "A graph-based evolutionary algorithm: Genetic network programming (GNP) and its extension using reinforcement learning," *Evol. Comput.*, vol. 15, no. 3, pp. 369–398, 2007.
[12] X. Li, S. Mabu, and K. Hirasawa, "A novel graph-based estimation of the distribution algorithm and its extension using reinforcement learning," *IEEE Trans. Evol. Comput.*, vol. 18, no. 1, pp. 98–113, Feb. 2014.
[13] K. Hirasawa, T. Eguchi, J. Zhou, L. Yu, and S. Markon, "A double-deck elevator group supervisory control system using genetic network programming," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 38, no. 4, pp. 535–550, Jul. 2008.
[14] X. Li and K. Hirasawa, "Continuous probabilistic model building genetic network programming using reinforcement learning," *Appl. Soft Comput.*, vol. 27, pp. 457–467, Feb. 2015.
[15] Y. Chen, S. Mabu, and K. Hirasawa, "A model of portfolio optimization using time adapting genetic network programming," *Comput. Oper. Res.*, vol. 37, no. 10, pp. 1697–1707, 2010.
[16] S. Mabu, C. Chen, N. Lu, K. Shimada, and K. Hirasawa, "An intrusion-detection model based on fuzzy class-association-rule mining using genetic network programming," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 41, no. 1, pp. 130–139, Jan. 2011.
[17] H. Katagiri, K. Hirasawa, J. Hu, and J. Murata, "Comparing some graph crossover in genetic network programming," in *Proc. SICE Conf.*, Aug. 2002, pp. 1263–1268.
[18] X. Li, W. He, and K. Hirasawa, "Genetic network programming with simplified genetic operators," in *Proc. Int. Conf. Neural Inf. Process.* Berlin, Germany: Springer, 2013, pp. 51–58.
[19] S. W. Wilson, "Classifier fitness based on accuracy," *Evol. Comput.*, vol. 3, no. 2, pp. 149–175, 1995.
[20] M. E. Pollack and M. Ringuette, "Introducing the tile-world: Experimentally evaluating agent architectures," in *Proc. AAAI*, 1990, pp. 183–189.
[21] T. Eguchi, K. Hirasawa, J. Hu, and N. Ota, "A study of evolutionary multiagent models based on symbiosis," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 36, no. 1, pp. 179–193, Feb. 2006.
[22] D. B. Fogel, "An introduction to simulated evolutionary optimization," *IEEE Trans. Neural Netw.*, vol. 5, no. 1, pp. 3–14, Jan. 1994.
[23] R. Poli, "Parallel distributed genetic programming," School Comput. Sci., Univ. Birmingham, Birmingham, U.K., Tech. Rep. CSRP-96-15, 1996.
[24] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. Eur. Conf. Genetic Program.*, 2000, pp. 121–132.
[25] K. Hirasawa, M. Ohbayashi, S. Sakai, and J. Hu, "Learning Petri network and its application to nonlinear system control," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 28, no. 6, pp. 781–789, Dec. 1998.
[26] R. Collin and R. Cipriani, "Dollo's law and the re–evolution of shell coiling," *Roy. Soc. London B, Biol. Sci.*, vol. 270, no. 1533, pp. 2551–2555, 2003.
[27] X. Li and G. Yang, "Artificial bee colony algorithm with memory," *Appl. Soft Comput.*, vol. 41, pp. 362–372, Apr. 2016.
[28] M. V. Butz and S. W. Wilson, "An algorithmic description of XCS," *Soft Comput.*, vol. 6, nos. 3–4, pp. 144–153, 2002.
[29] H. Williams, W. N. Browne, and D. A. Carnegie, "Learned action SLAM: Sharing slam through learned path planning information between heterogeneous robotic platforms," *Appl. Soft Comput.*, vol. 50, pp. 313–326, Jan. 2017.
[30] M. V. Butz, D. E. Goldberg, and P. L. Lanzi, "Gradient descent methods in learning classifier systems: Improving XCS performance in multistep problems," *IEEE Trans. Evol. Comput.*, vol. 9, no. 5, pp. 452–473, Oct. 2005.
[31] M. V. Butz, "XCSJava 1.0: An implementation of the XCS classifier system in Java," Illinois Genet. Algorithms Lab., UIUC, Champaign, IL, USA, Tech. Rep. 2000027, 2000.
[32] R. Poli, W. Langdon, and N. McPhee. (2008). *A Field Guide to Genetic Programming (With Contributions by JR Koza)*. [Online]. Available: http://lulu.com
[33] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.
[34] S. Luke, "ECJ then and now," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2017, pp. 1223–1230.

XIANNENG LI received the B.E. degree from Nanjing University, China, in 2008, and the Ph.D. degree from Waseda University, Japan, in 2013.

From 2013 to 2015, he was a JSPS Post-DOCTORAL Research Fellow and a Researcher (Lecturer) with Waseda University. He is currently an Associate Professor with the Faculty of Management and Economics, Dalian University of Technology, China. He has authored over 40 papers in international journals and conferences. His current research interest includes evolutionary computation, transfer learning and their applications on control problems, data mining, and e-commerce.

Dr. Li is currently an Editorial Board Member of *Applied Soft Computing* journal, a referee of over 10 international journals, and a PC member of over 20 international conferences.

HUIYAN YANG received the B.E. degree from Inner Mongolia University, China, in 2016. She is currently pursuing the M.E. degree with the Faculty of Management and Economics, Dalian University of Technology, China. Her current research interest includes evolutionary computation and its applications on control problems.

MEIHUA YANG received the B.E. degree from the University of Science and Technology Beijing, China, in 2017. She is currently pursuing the M.E. degree with the Faculty of Management and Economics, Dalian University of Technology, China. Her current research interest includes machine learning and its applications on data mining.